ParaDualMiner: An Efficient Parallel Implementation of the DualMiner Algorithm

Roger M. H. Ting and James Bailey and Kotagiri Ramamohanarao

Department of Computer Science and Software Engineering
The University of Melbourne, Australia

Abstract. Constraint based mining finds all itemsets that satisfy a set of predicates. Many constraints can be categorised as being either monotone or antimonotone. Dualminer was the first algorithm that could utilise both classes of constraint simultaneously to prune the search space. In this paper, we present two parallel versions of DualMiner. The ParaDualMiner with Simultaneous Pruning efficiently distributes the task of expensive predicate checking among processors with minimum communication overhead. The ParaDualMiner with Random Polling makes further improvements by employing a dynamic subalgebra partitioning scheme and a better communication mechanism. Our experimental results indicate that both algorithms exhibit excellent scalability.

1 Introduction

Data mining in the presence of constraints is an important problem. It can provide answers to questions such as "find all sets of grocery items that occur more than 100 times in the transaction database and the maximum price of the items in each of those sets is greater than 10 dollars". To state our problem formally, we denote an item as i. A group of items is called an itemset, denoted as S. The list of items that can exist in our database is denoted as $I = \{i_1, i_2, \ldots, i_n\}$, $S \subseteq I$. The constraints are a set of predicates $\{P_1, P_2, \ldots, P_n\}$ that have to be satisfied by an itemset S. Constraint based mining finds all sets in the powersets of I that satisfy $P_1 \wedge P_2 \wedge \ldots \wedge P_n$.

Many constraints can be categorised into being either monotone or antimonotone constraints [8, 10]. There exist many algorithms that can only use one of them to prune the search space. There are also algorithms that can mine itemsets using these two constraint categories in a sequential fashion (e.g. [13]). DualMiner was the first algorithm able to interleave both classes of constraint simultaneously during mining [4]. Nevertheless, the task of finding itemsets that satisfy both classes of constraint is still time consuming. High performance computation offers a potential solution to this problem, provided that an efficient parallel version of DualMiner can be constructed.

Contributions: In this paper, we introduce two new parallel algorithms which extend the original serial DualMiner algorithm [4]. We have implemented our algorithms on a Compaq Alpha Server SC machine and they both show excellent scalability. To the best of our knowledge, our algorithms are the first parallel

algorithms that can perform constraint-based mining using both monotone and antimonotone constraints simultaneously.

2 Preliminaries

We now provide a little background on the original DualMiner algorithm [4]. Formally, given itemsets M,J and S, a constraint is *antimonotone* if

$$\forall S, J : ((J \subseteq S \subseteq M) \wedge P(S)) \Rightarrow P(J)$$

One of the most widely cited antimonotone constraints is $support(S) > c$. A constraint is *monotone* if

$$\forall S, J : ((S \subseteq J \subseteq M) \wedge Q(S)) \Rightarrow Q(J)$$

Monotone constraints are the opposite of antimonotone constraints. Therefore, a corresponding example of a monotone constraint is $support(S) < d$. A conjunction of antimonotone predicates is antimonotone and a conjunction of monotone predicates is monotone [4]. Therefore, many itemset mining problems involving multiple constraints can be reduced to looking for all itemsets that satisfy a predicate of the form $P(S) \wedge Q(S)$. Even though some constraints are not exactly monotone or antimonotone constraints, previous research indicates that they can be approximated to be either monotone or antimonotone if some assumptions are made [10]. According to previous work, the search space of all itemsets forms a *lattice*. Given a set I with n items, the number of elements in the lattice is 2^n. This is equal to the number of elements in the powerset of I, which is denoted as 2^n. By convention, the biggest itemset is at the bottom of this lattice and the smallest itemset will always be at the top. Beside that, our search space also forms a *Boolean algebra* with maximal element B and minimal element T. It has the following properties (i)$X \in \Gamma$ (ii) $B = \bigcup X$ which is the bottom element of Γ (iii) $T = \bigcap X$ which is the top element of Γ (iv) for any $A \in \Gamma$, $\overline{A} = B \backslash A$. A *subalgebra* is a collection of elements $\subseteq 2^n$ closed under $\bigcap$ and $\bigcup$. The top and bottom element of the algebra is sufficient to represent all the itemsets in between them. If the top and bottom elements satisfy both constraints, the monotone and antimonotone properties guarantee that all itemsets in between them will satisfy both constraints. A *good subalgebra* is a subalgebra which has top and bottom elements that satisfy both the antimonotone and monotone constraints.

Overview of DualMiner DualMiner builds a dynamic binary tree when searching for all good subalgebras. A tree node represents a subalgebra, but not necessarily a good subalgebra. Each tree node τ consists of three item lists which are (i) $IN(\tau)$ representing all the items that must be in the subalgebra and the top element of current subalgebra, T, (ii) $CHILD(\tau)$ representing all the items that have not been apportioned between $IN(\tau)$ and $OUT(\tau)$ and (iii) $OUT(\tau)$ representing all the items that cannot be contained in the current subalgebra. Because

our search space forms a Boolean algebra, $\overline{OUT}$ represents the bottom element of the current subalgebra, B. Note that $\overline{OUT(\tau)} = \{IN(\tau) \cup CHILD(\tau)\}$.

When DualMiner starts, it will create a root node with $IN(\alpha)$ and $OUT(\alpha)$ empty . It will start checking the top element of the current subalgebra first. If the top element does not satisfy the antimonotone constraint, every itemset below it will not satisfy the constraint too. Therefore, we can eliminate the subalgebra. If it satisfies the antimonotone constraint, DualMiner will check all the itemsets below T in the subalgebra using the antimonotone constraint. Each is of the form $IN \cup \{X\}$, where X is an item from the $CHILD$ item list. If all itemsets $IN \cup \{X\}$ satisfy the constraint, no item list will be altered. If an itemset does not satisfy the constraint, X will be put into OUT item list. This effectively eliminates the region that contains that item.

Next, the algorithm will apply the monotone constraint on B of the current subalgebra. If the maximal itemset fails, the algorithm eliminates the current subalgebra immediately. If it does not fail, DualMiner will start checking all the itemsets one level above the current bottom itemset using the monotone constraint. Each is of the form $\overline{OUT \cup \{X\}}$, where X is an item from the $CHILD$ item list. If all itemsets $\overline{OUT \cup \{X\}}$ satisfy the constraint, no item list will be altered. If an itemset does not satisfy the constraint, X will be put into the IN item list. This eliminates the region that does not contain that item.

The pruning process will continue until no pruning can be done. At the end of the pruning phase, the top itemset must satisfy the antimonotone constraint and the bottom itemset must satisfy the monotone constraint. If the top itemset also satisfies the monotone constraint and the bottom itemset also satisfies the antimonotone constraint, we have found one good subalgebra. If this is not the case, DualMiner will partition the subalgebra into two halves. This is done by firstly creating two child tree nodes and picking an item from the $CHILD$ itemset and inserting it into the IN of one child and OUT of another child. The algorithm will mark the current parent node as visited and proceed to the child nodes. The process is repeated until all nodes are exhausted.

3 Parallel DualMiner

DualMiner does not prescribe specific constraints that have to be used. The antimonotone constraint and the monotone constraint are two types of predicates over an itemset or oracle functions that return true or false. To simplify our implementation, we will use $support(S) > C$ as our antimonotone constraint and $support(S) < D$ as our monotone constraint. $C \leq D$. We represent the database with a series of bit vectors. Each bit vector represents an item in the database. The support count of an itemset can be found by performing a bitwise AND operation on the bit vector of each item in the itemset. This approach has been used by many other algorithms[5, 9]. We represent the IN, $CHILD$ and OUT itemlist in each node as three bit vectors. Each item is represented as 1 bit in each of the three bit vectors. The position of the bit will indicate the item id of the item.

ParaDualMiner with Simultaneous Pruning In the original DualMiner paper, it was observed that the most expensive operation in any constraint based mining algorithm is the oracle function. Therefore, we can achieve great performance gain if we can distribute the call to the oracle function evenly among different processors. We notice that after DualMiner verifies that the top element of a subalgebra satisfies the antimonotone constraint, it will check whether all the elements one level below the top element satisfies the antimonotone constraint. Each of them is of the form $IN \bigcup \{X\}$, where X is any item from the $CHILD$ itemset. Since each oracle function call is independent, it is possible to partition the $CHILD$ item list and perform the oracle function call simultaneously. e.g. Given the following transaction database: Transaction 1={A,B,C,D}, Transaction 2 = {A,B,C} and suppose our constraints are $support(S) > 1$ and $support(S) < 3$, the execution of the algorithm is illustrated in figure 1. Before partitioning the

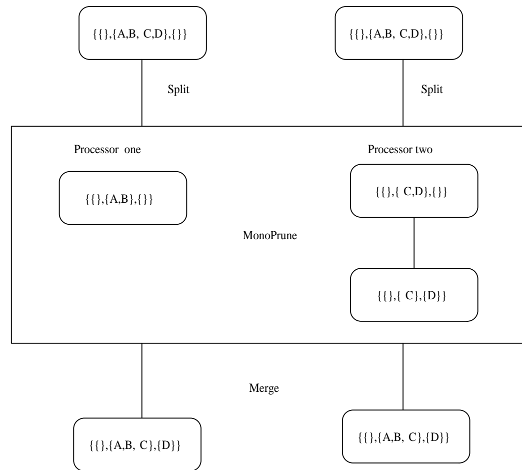

Fig. 1. ParaDualMiner with Simultaneous Pruning

$CHILD$ item list, all the processors will have the same IN,$CHILD$ and OUT item list. After the parallel algorithm distributes the antimonotone constraint checking among different processors, any itemset of the form $IN \bigcup \{X\}$ such as $\{D\}$, that does not satisfy the antimonotone constraint, will lead to an item being inserted into the OUT item list in order to prune away that part of the search space. Therefore, at the end of the simultaneous antimonotone checking, the item lists that will be altered are the $CHILD$ and OUT item list.

Before proceeding, we must merge the search space that has not been pruned away using the antimonotone constraint. It only has to perform a bitwise boolean OR operation on the $CHILD$ and OUT item lists of all processors. This will give us the global result based on the individual processor pruning process. This

simplification is the direct result of us choosing the bit vector as our item list representation. The merging operation can be done between all processors using the MPI_Allreduce function, with boolean OR as the operator.

A similar process can be applied when we are pruning the search space using the monotone constraint. The difference is the algorithm is partitioning and merging the IN and $CHILD$ item lists instead of the OUT and $CHILD$ itemlists. The partitioning operation is entirely a local operation.There is no message passing involved. Each processor will check the number of items in the $CHILD$ item list, the number of processors and its own rank to consider which part of the $CHILD$ itemlist to be processed. If it can divide the number of items evenly, each processor will perform an equal amount of oracle calls. However, this will happen only if the number of processors is a perfect multiple of the number of items in the $CHILD$ item list. If the algorithm cannot divide the $CHILD$ item list evenly, the algorithm will distribute the residual evenly to achieve optimal load balancing. Therefore, the maximum idle time for processors each time the algorithm distributes the task of oracle function call will be T_{oracle}.

ParaDualMiner with Random Polling There are a number of parallel frequent pattern miners that use the concept of candidate partitioning and migration to distribute task among processors (e.g. [7, 1]). We can see similar behaviour in DualMiner. Whenever DualMiner cannot perform any pruning on the current subalgebra using both constraints, DualMiner will split the current subalgebra into two halves by splitting the tree node. This node splitting operation is essentially a divide-and-conquer strategy. No region of the search space has been eliminated in the process. Therefore, the algorithm permits an arbitrary amount splitting of subalgebras subject to the condition that they are to be evaluated later. The number of splits permitted is equal to the number of items in the $CHILD$ item list. This intuition gives us the simplest form of a parallel subalgebra partitioning algorithm.

In the 2 processor case, the original search space is partitioned into two subalgebras. Both processors can turn off 1 bit in the $CHILD$. One puts it in the IN item list by turning on the similar bit in the IN bit vector. Another processor will put it in the OUT item list by turning on the similar bit. The two processors search two disjoint search spaces without any need for communication. After the original algebra has been partitioned, each processor will simultaneously run Dualminer locally to find itemsets that satisfy both constraints. Since our search space can be partitioned up to the number of items in the $CHILD$ itemlist, this strategy can be applied to cases with more than two processors. The number of processors that are needed must be 2^n, where n is the number of times the splitting operation has been performed. The partitioning operation is a local process. Each processor will only process one of the subalgebras according to its own rank. There is no exchange of messages.

This algorithm will only achieve perfect load balancing if the two nodes contain equal amounts of work. This is unlikely because it is unlikely that the search space of each processor is even. One of the processors may terminate

earlier than the rest of processors. Without any dynamic load balancing, the processor will remain idle throughout the rest of the execution time. This leads to poor load balancing and longer execution time. To overcome this problem, we can view a node as a job parcel. Instead of letting a free processor stay idle throughout the execution time, the new algorithm can delegate one of the nodes to an idle processor to obtain better load balancing.

There are two ways to initiate task transfer between processors. They are sender-initiated and receiver-initiated methods[6]. Our study indicated that the receiver-initiated scheme outperformed the sender-initiated scheme. The reason for this is that the granularity of time when a processor is idle in the receiver-initiated scheme is large. DualMiner spends most of its time in the pruning process. Therefore, the time a processor takes before it splits a node can be very long. If a processor terminates very early at the start of its own pruning process, it has to passively wait for another processor splits a node and sends it. This greatly decreases the work done per time unit which leads to poor speedup. Instead of being passive, the idle processor should poll for a job from a busy processor. DualMiner can split a node anywhere, provided that we do the splitting and job distribution properly. e.g. Given the transaction database Transaction 1 = {A,B,C,D}, Tranaction 2= {C,D} and the constraint is $support(S) > 1$ and $support(S) < 3$,the set of itemsets that satisfies both constraints is $\{\{C\}, \{D\}, \{C, D\}\}$.

The original search space will firstly be split into two subalgebras as shown in figure 2. Since itemset $\{A\}$ is infrequent, processor one that processes the left node will finish earlier. Processor two that processes right node could be still within one of the pruning functions. Instead of staying idle, processor one should then poll for a job from processor two.

Suppose processor two is pruning the search space using the antimonotone constraint. This implies that the top element of the current subalgebra such as $\{\}$ has already satisfied the antimonotone constraint. Otherwise, this subalgebra would have been eliminated. Therefore, while it is evaluating all the elements one level below the top element, it can check for an incoming message from processor one. Suppose it finds that processor one is free after evaluating itemset $\{B\}$, it can split the subalgebra and send it to processor one as shown in figure 2. In this case, the subalgebra is further split into two smaller subalgebras and can be distributed between these two processors. When the algorithm is pruning the search space using the antimonotone constraint, the IN itemset must have already satisfied the antimonotone constraint. Therefore, in this example, if processor two continues pruning using the antimonotone constraint, processor two should pick the right node. Likewise, if the algorithm is pruning the search space using monotone constraint, the sender processor should pick the left node. Suppose that processor two has already split a node and there is already a node or subalgebra that is yet to be processed. The algorithm should send that node to the idle processor instead of splitting the current one that it is working on. This is because the size of the subalgebra that is yet to be processed is equal to or greater than the size of the current subalgebra, if we are using a depth first

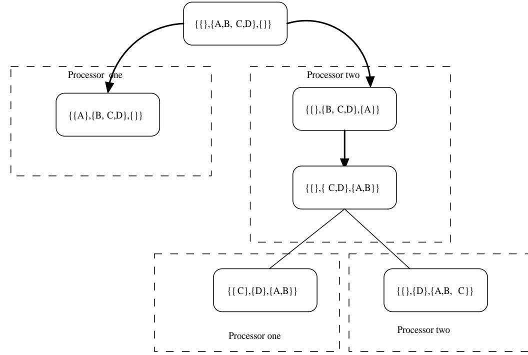

Fig. 2. Subalgebra Partitioning with Random Polling

or breadth first traversal strategy .

There are mainly two approaches to extend this algorithm to multiple processors. They are categorised into decentralised and centralised schemes [15]. To simplify our implementation, we have adapted the master-slave scheme. The master processor acts as an agent between all the slave processors. Each slave processor will work on the subalgebras that are assigned to it simultaneously. However, it will anticipate an incoming message from the master. Whenever a slave processor runs out of jobs locally, it will send a message to the master. The master processor will then poll for a job from a busy processor. Therefore the master has to know which processors are busy and which are idle.

For this purpose, we keep a list of processors that are busy and idle. The list can be efficiently represented as a bit vector. The bit position of the vector will then be the rank of the processor. A busy processor will be denoted as 1 in the bit vector. A free processor will be denoted as 0 in the bit vector. Whenever the master receives a message from the processor X, it will initialise bit X to zero. It will then select a processor to poll for job. There are various way to select a processor. A random selection algorithm has been found to work very well in many cases. Also, there is previous work that analyses the complexity of this kind of random algorithm [11, 12, 11]. Therefore, we have used this algorithm in our implementation. The master will randomly generate a number between 0 and $n-1$, where n is the number of processors. It will then send a free message to the selected processor to poll for a job. If the selected slave processor does not have any job, it will send a free message to the master processor. The master processor will then mark it as free and put it into a free CPU queue. It will continue polling until a job message is replied to it. It will then send the job to a free processor. The slave processors can detect incoming messages using the MPI_Iprobe function. The termination condition is when the number of free processors in the list is equal to the number of processors available. This implies that there is no outstanding node to be processed. The master processor will

then send a termination message to all the slave processors.

4 Experiments

We implemented both serial and parallel versions of DualMiner on a 128-processor Unix Cluster.The specification of the machine is 1 Terabyte of shared file storage, 128 Alpha EV68's at 833 MHz processor, a Quadrics interconnect which has 200 Megabyte/sec bandwidth and 6 milliseconds latency and 64 Gigabytes of memory. We used databases generated from the IBM Quest Synthetic data generator. The datasets generated from it are used in various papers [4, 2, 14]. The number of transactions is 10000. The dimension of the dataset is 100000, which means maximum number of distinct items in the transaction database is 100000. The length of a transaction is determined by a Poisson distribution with a parameter, average length. The average length of transactions is 10. We also scaled up the dimension of dataset by doubling the average length of transactions. This is because the computing resources demanded is significantly higher if the dataset is dense [3]. For our purpose, we define a dataset with an average transaction length of 10 to be sparse and one with an average length of 20 to be dense.

In the Random Polling version of ParaDualMiner, the master node only acts as an agent between all the slave processors. It does not run the DualMiner like the slave processors. At the start of algorithm, the original algebra is partitioned into 2^n part. This means this version of ParaDualMiner can only accept 2^n processors for the slaves and one additional processor for the master. Therefore, we studied our algorithm with 2,3,5,9,17 processors.

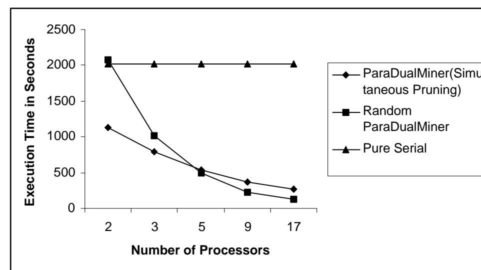

Fig. 3. support between 0.015 and 0.03 percent and sparse dataset

Results As shown in figure 3 and figure 4, the execution time of ParaDualMiner with Random Polling is almost identical to the serial version of DualMiner even though it is using 2 processors. The reason is that the master in Random ParaDualMiner does not perform any task, besides acting as an agent between all the

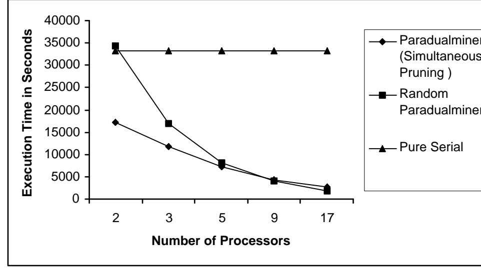

Fig. 4. support between 0.015 and 0.03 percent and dense dataset

slave processors. Therefore, in the 2 processors case, there is only one processor working on the original algebra. When there are 3 processors, there will be two slave processors that work on the subalgebra distributed to them. Since the master will always poll for job from the busy processor and it is always possible to split work, the load balancing is excellent. Beside that, every communication is point to point communication and the message is relatively small. This leads to almost perfectly linear speedup after 2 processors. We also observe that there is super linear speedup in some cases. This is due to better memory usage. Whenever a processor has relatively more nodes to process, the nodes will migrate to other processors with less work load. This distributes the memory requirement among all the processors.

From figure 3 and figure 4, we can see that ParaDualMiner with Simultaneous Pruning is not as scalable as Random ParaDualMiner. The reason is that whenever there is n processors, there will be exactly n processors that will split the most computational intensive part of the algorithm which is the oracle function call. However, the algorithm will only achieve perfect load balancing if the number of items in the $CHILD$ itemlist is a perfect multiple of the number of processors. As the number of processors increases, the chances of getting a perfect multiple of the number of processors decreases. This implies the chance of having some processors stay idle for one oracle function call becomes larger. This may cause many processors to become idle too often, which impairs the parallelism that can be achieved by this algorithm. Also, the algorithm only parallelises the oracle function call. Furthermore, there is a need to have all-to-all communication to merge all the result of pruning. This is much more expensive than point-to-point communication in ParaDualMiner with Random Polling.

5 Conclusion

We have proposed two parallel algorithms for mining itemsets that must satisfy a conjunction of antimonotone and monotone constraints. There are many serial or parallel algorithms that take advantage of one of these constraints. However, both of our parallel algorithms are the first parallel algorithms that

take advantage of both constraints constraints simultaneously to perform constraint based mining. Both algorithms demonstrate excellent scalability. This is backed by our experimental result. We are currently investigating the scalability of both algorithms using hundreds of processors. Also, we are investigating how ParaDualMiner performs if we use other type of constraints. We believe that both parallel algorithms should perform well, because there is no reliance on the underlying nature of the constraints.

References

1. R. Agrawal and J. C. Shafer. Parallel mining of association rules. *IEEE Trans. On Knowledge And Data Engineering*, 8:962–969, 1996.
2. R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of VLDB'94*, pages 487–499, 1994.
3. R. Bayardo, R. Agrawal, and D. Gunopulos. Constraint-based rule mining in large, dense databases. In *Proceedings of ICDE'99*, pages 188–197, 1999.
4. C. Bucila, J. Gehrke, D. Kifer, and W. White. Dualminer: a dual-pruning algorithm for itemsets with constraints. In *Proceedings of ACM SIGKDD'02*, pages 42–51, 2002.
5. D. Burdick, M. Calimlim, and J. Gehrke. Mafia: A maximal frequent itemset algorithm for transactional databases. In *Proceedings of ICDE'01*, pages 443–452, 2001.
6. D. L. Eager, E. D. Lazowska, and J. Zahorjan. A comparison of receiver-initiated and sender-initiated adaptive load sharing. In *Proceedings of ACM SIGMETRICS'85*, pages 1–3, 1985.
7. E. H. Han, G. Karypis, and V. Kumar. Scalable parallel data mining for association rules. In *Proceedings of SIGMOD'97*, pages 277–288, 1997.
8. R. T. Ng, L. V. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained association rules. In *Proceedings of SIGMOD'98*, pages 13–24, 1998.
9. S. Orlando, P. Palmerini, R. Perego, and F. Silvestri. Adaptive and resource-aware mining of frequent sets. In *Proceedings of ICDM'02*, page 338, 2002.
10. J. Pei, J. Han, and L. Lakshmanan. Mining frequent item sets with convertible constraints. In *Proceedings of ICDE'01*, pages 433–442, 2001.
11. P.Sanders. A detailed analysis of random polling dynamic load balancing. In *Proceedings of ISPAN'94*, pages 382–389, 1994.
12. P.Sanders. Asynchronous random polling dynamic load balancing. In *Proceedings of ISAAC'99,*, page 39, 1999.
13. L. D. Raedt and S. Kramer. The levelwise version space algorithm and its application to molecular fragment finding. In *Proceedings of IJCAI'01*, pages 853–8622, 2001.
14. O. Zaiane, M. El-Hajj, and P. Lu. Fast parallel association rule mining without candidacy generation. In *Proceedings of ICDM'01*, pages 665–668, 2001.
15. M. Zaki, W. Li, and S. Parthasarathy. Customized dynamic load balancing for a network of workstations. *Journal of Parallel and Distributed Computing*, 43(2):156–162, 1997.

This article was processed using the LaTeX macro package with LLNCS style